

# ICS/CSE 141: Programming Languages

## Unification & Prolog

School of Information and Computer Science  
University of California, Irvine

Chris Stork  
cstork@ics.uci.edu

## Overview

1. Rules and Backtracking
2. Unification
3. Prolog

# Rules & Backtracking

All examples refer to the file `infer.scm`

## Rules

We want to be able to say:

```
(and (parent ?x ?y) (male ?x)) ==> (father ?x ?y)
(and (parent ?y ?x) (male ?x)) ==> (son ?x ?y)
```

### Rules as DB entries

```
(←- (father ?x ?y)                                     ⇐ head of the rule
    (and (parent ?x ?y) (male ?x)))                    ⇐ body of the rule

(←- (son ?x ?y)
    (and (parent ?y ?x) (male ?x)))
```

### Famous Syllogism by Socrates:

1. All men are mortal.
2. Socrates is a man.
3. Therefore Socrates is mortal.

## Backtracking (depth-first)

bt-ex test from `infer.scm`. Draw the tree!

```
(define bt-ex-rules
  '(
    (<- (a)
        (and (b) (c) (d)))
    (<- (a)
        (and (e) (f)))
    (<- (b)
        (f))
    (e)
    (<- (f)      ; Another way of stating (f) as a fact;
        (and)) ; works since (and) is always true!
    (<- (a)
        (f))
  ))

(set! *db* bt-ex-rules)

(test bt-ex
  ((query-infer '(and (a) (e)))
   (( ) ( ) ( ))
   )
  )
```

## Recursion & Infinite Search Space

Replace the last rule

```
( ← ( a ) ( f ) )
```

by

```
( ← ( a ) ( and ( f ) ( a ) ) )
```

Lazy evaluation would return infinitely many solutions.

What happens if we make the last rule the first rule?

# Unification

## Beyond Matching

Draw (father homer bart) example!

We need **unification** to “match” our goal against heads of rules!

## Unification

Unification is like our `match` on steroids. It can “match” terms that **both** contain logical variables.

First we will take look at unification in isolation as presented in Mitchell, Chapter 15. This should then give us the “confidence” that our implementation works.

## Terms

Unification operates on **terms**, e.g.,

$$f(x, a), f(g(x), y), h(u)$$

Terms consist of

- **variables:**  $x, y, z, \dots$  (our “logical variables”)
- **function symbols:**  $f, g, h, \dots$  (our “relations”)
- parenthesis and comma

Terms are inductively defined as

- a variable
- if  $f$  is an  $n$ -ary function symbol and  $t_1, \dots, t_n$  are terms, then  $f(t_1, \dots, t_n)$  is a term

Constants (0-ary functions) are written as  $a, b, c, \dots$  (our “objects” or  $(\mathbb{F})$ )

## Example: Unification

Examples of Unification:

- $a = a \Rightarrow \text{yes}, [ ]$
- $f(b) = f(b) \Rightarrow \text{yes}, [ ]$
- $f(a) = f(b) \Rightarrow \text{no}$
- $f(x) = f(2a) \Rightarrow \text{yes}, [ ?x=a ]$
- $f(x) = f(y) \Rightarrow \text{yes}, [ ?x=?y ]$
- $f(x, y) = f(a) \Rightarrow \text{no}$
- $f(x, y) = f(a, x) \Rightarrow \text{yes}, [ ?x=a, ?y=a ]$

A variable can only have **one** value for a given unification to succeed.

## Unification Algorithm

Given a **set of equations** to unify, **repeat**

1.  $f(s_1, \dots, s_n) = f(t_1, \dots, t_n)$     replace by  $s_1 = t_1, \dots, s_n = t_n$
2.  $f(s_1, \dots, s_n) = h(t_1, \dots, t_n), f \neq h$     halt with failure
3.  $x = x$     delete equation
4.  $t = x, t$  not a variable    replace by  $x = t$
5.  $x = t, x$  not in  $t$  and  $x$  elsewhere    substitute  $[x/t]$  everywhere
6.  $x = t, x$  in  $t$  and  $x \neq t$     halt with failure

**until failure or no match anymore!**

This is proven to terminate and find the **most general unifier (MGU)**. (So if we implement this we can be sure to do the “right thing”.)

## More Examples

Example: Consider  $\{f(x, a) = f(b, y)\}$  for unification

- Use (1) to get  $\{x = b, a = y\}$
- Use (4) to get  $\{x = b, y = a\}$
- Nothing else applicable, unifies using  $[x/b]$  and  $[y/a]$

Example: Consider  $\{f(x, a) = f(g(z), y), h(u) = h(d)\}$

- Use (1) on first to get  $\{x = g(z), a = y, h(u) = h(d)\}$
- Use (1) on third to get  $\{x = g(z), a = y, u = d\}$
- Use (4) on second to get  $\{x = g(z), y = a, u = d\}$
- Nothing else applicable, unifies using  $[x/g(z)]$ ,  $[y/a]$ ,  $[u/d]$

## More Examples

Consider  $\{f(x, a) = f(g(z), y), h(x, z) = h(u, d)\}$

- Use (1) on first to get  $\{x = g(z), a = y, h(x, z) = h(u, d)\}$
- Use (5) with first to get  $\{x = g(z), a = y, h(g(z), z) = h(u, d)\}$
- Use (1) on third to get  $\{x = g(z), a = y, g(z) = u, z = d\}$
- Use (5) with fourth to get  $\{x = g(d), a = y, g(d) = u, z = d\}$
- Use (4) on second to get  $\{x = g(d), y = a, g(d) = u, z = d\}$
- Use (4) on third to get  $\{x = g(d), y = a, u = g(d), z = d\}$
- Nothing else applicable, unifies using  $[x/g(d)], [y/a], [u/g(d)], [z/d]$

## Our Unification Implementation

Our implementation mirrors Apt's, except that we put the equations with a single logical variable on the left-hand-side in our `frame`.

```
(define (query-infer q)
  (query-infer-rec q '()))

;; Like query-rec except that it calls the infer versions and
;; infer-rec-db calles unify...
(define (query-infer-rec q frame)
  (case (car q)
    ((and) (query-infer-and (cdr q) frame))
    ((or) (query-infer-or (cdr q) frame))
    ((not) (query-infer-not (cadr q) frame))
    (else (infer-rec-db q frame))))
```

Looks like query! Difference lies in `infer`. Look at code: first `simpler-but-wrong-infer` and then `infer`.

## Interesting Stuff

Look at `append-to` test case. Surprise, we can reason “backwards”!

(We can also deal with lists and recursion in our rules. How come this works? Demonstrate improper list syntax for `define` and look at `unify`'s implementation.)

There is many more interesting extensions to our implementation:

- lazy evaluation with `delay` and `force` (approximating Prologs real behavior)
- add Scheme expressions to be allowed as terms in rules

# Prolog

# Prolog

We just constructed the core of a programming languages called **Prolog!**

Stands for **programming in logic**

Not based on abstract machines or mathematical functions

But on a version of **predicate logic** instead

Prolog is a complete programming language with

- arithmetic
- lists
- special control facilities, most notably *cut*

## How we differ from Prolog

- Syntax:
  - variable names: Prolog uses capitalized names for logical variables
  - facts written as: `happy .`
  - implications as: `happy :- moreFood .` (called *clauses*)
- we use Scheme lists instead of Prologs special `[ H | T ]` lists

Prolog interactive queries:

```
append( [ ] , Xs , Xs ) .
```

```
append( [ X | Xs ] , Ys , [ X | Zs ] ) :- append( Xs , Ys , Zs ) .
```

```
> append( [ 1 , 2 ] , A , [ 1 , 2 , 3 , 4 ] ) ?
```

```
A = [ 3 , 4 ] .
```

## Logic Programming Languages

It's not the same as Prolog!! Much has changed since then. You only got a taste of what's out there.

### Check out:

- Jess (Lisp & expert system (Rete algorithm) written in Java)
- Mercury (descendent of Prolog with types and more)
- Kanren (declarative logic programming system embedded in Scheme!)
- Constraint-based programming ...

## What makes a language declarative

Functions are equalities, we just chose to read them one way! Take any of our purly functional examples, e.g. `app`.

Pure functions, rules, and relations define, or better, **declare** relationships!

# List of Slides

- 1 Lecture 7
- 1 Title
- 2 Overview
- 3 Rules & Backtracking
- 4 Rules
- 5 Backtracking (depth-first)
- 6 Recursion & Infinite Search Space
- 7 Unification
- 8 Beyond Matching
- 9 Unification
- 10 Terms
- 11 Example: Unification
- 12 Unification Algorithm
- 13 More Examples
- 14 More Examples
- 15 Our Unification Implementation
- 16 Interesting Stuff
- 17 Prolog
- 18 Prolog
- 19 How we differ from Prolog
- 20 Logic Programming Languages
- 21 What makes a language declarative